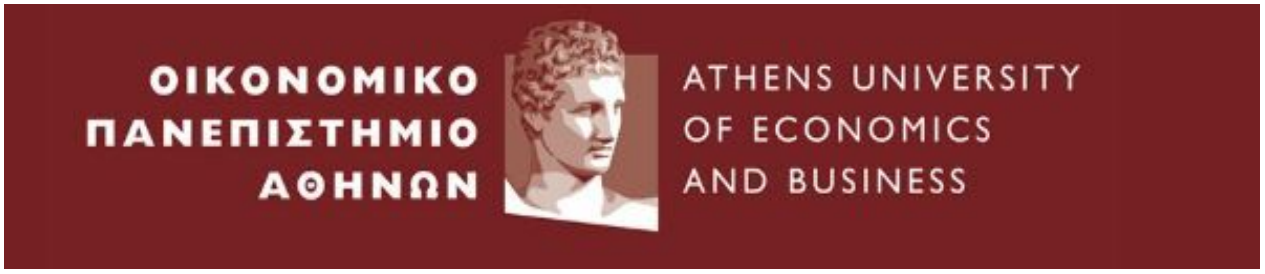


AUEB M.S.c in Data Science (part-time)

Text Analytics: Homework 4: Report

Team members:

Kaplanis Alexandros (P3351802)

Politis Spiros (P3351814)

Proimakis Manos (P3351815)

1. SETUP & EXECUTION

Please refer to the README.md for instructions.

This assignment is based on two different *Jupyter Notebooks*, namely:

- **assignment-4-training.ipynb**: contains all the code and relevant outcomes of the RNN training process.
- **assignment-4-inference.ipynb**: contains all the code that loads the trained and tuned RNN and performs inference on it.
- **assignment-4-elmo.ipynb:** contains all the code that loads the elmo pretrained model and evaluation

2. INTRODUCTION

We opted for reusing the dataset that was employed in our last assignment, namely the Twitter Sentiment dataset which has already been downloaded from Kaggle for the Sentiment140 project. We have already performed significant work on data preparation and word embeddings, therefore we consider our previous effort as a good baseline from which to start. As a reminder, the datasets targets 2 classes; 0 for negative and 4 for positive, which are equiprobable with 800,000 records per class.

2.1 SCOPE & GOAL

In the current iteration our goal is to train a *RNN* implemented in *Keras* and tune the model to achieve a good prediction score as well as be sufficiently generalizable for inference. In our previous attempt we have chosen a tuned *MLP* model using *TF/IDF* vectors, which achieved 0.82 *f1 score* on the unseen data. The interesting thing, given the good scores we achieved using both the *MLP* and the *Logistic Regression*, would be to examine whether the *RNN* would perform better than the other two models.

2.2 EMBEDDINGS

2.2.1 FastText

This time around, we opted for using *FastText* embeddings. Having already experimented with TF/IDF vectorization and *Word2Vec* embeddings we decided to try *FastText*, an extension to *Word2Vec* proposed by *Facebook* in 2016. Instead of vectorizing individual words, *FastText* breaks words into several n-chars (sub-words), providing embeddings based on those n-chars. The immediate effect of this is that words that do not appear regularly in the vocabulary can now be properly represented, since it is highly likely that some of their n-chars also appears in other words.

2.2.2 ELMO

In this iteration we also experimented with *ELMO* embeddings. We have used the pretrained *ELMO* model from *Google* to create context aware embeddings which we retrained as we have a very big pool of tweets. *ELMo* is a deep contextualized word representation that models both (1) complex characteristics of word use (e.g., syntax and semantics), and (2) how these uses vary across linguistic contexts (i.e., to model polysemy). These word vectors are learned functions of the internal states of a deep bidirectional language model (*biLM*), which is pre-trained on a large text corpus. They can be easily added to existing models and significantly improve NLP problems. *ELMo* embeddings have a bigger size of 1024 in comparison with the *FastText* embeddings which had a size of 300 and it required much more time to train, as we used a batch of 50 inputs per iteration because we didn't have a GPU with enough memory for higher batches..

3. RNN Architecture

In order to have a common API for the implementation of different flavors of the *RNN* architecture (with *GRU* / *LSTM* cells, linear / deep self-attention etc.), we implemented a bespoke *Python* factory class which allows for the sequential building of an *RNN*. Since we had to account for self-attention mechanics, we opted for the *Keras* functional API so as to have a common infrastructure for the creation of an *RNN*. The custom class does not claim to be a complete wrapper for building *RNNs* with *Keras*, it merely implements such methods as to allow us to implement the *RNNs* required for this assignment.

Initially, we used *GRU* cells which we were able to train in much less time than the *LSTM* cells. Training was attempted on a machine with a reasonably powerful GPU (*Nvidia GTX1060*), since training performance is greatly increased using the *tensorflow-gpu* / *CUDA* implementation. Indeed, attempting to train a *RNN* on the CPU is painfully slow.

Subsequently, we proceeded with implementing layer(s) of *LSTM* cells, as required by the assignment. In terms of training performance, we anticipated an increased demand in computing power, based on the fact that *LSTM* units involve quite a few more tensor operations, a notion which was confirmed by our experiments.

As a first step we decided to have a look on which parameters play a key role in the accuracy of our model. We didn't want to use any automation on this step, as it is very time consuming, and it would be very important to tune only the parameters that would help increase the efficiency. On the other hand, manual tuning does not allow us to see every possible combination of the parameters. As a result we experimented by tuning one parameter at a time, noted which were the best performing values and then moved on to the next parameter. This way we got a good grasp on which parameter values work for our model which will be further tested in the hyperparameter tuning phase with *Talos*.

The properties of the *GRU* cells along with the dense layer were determined through parameter tuning, but initially we have run many exploratory trials to limit our tuning parameters, based on intuition about the key parameters that might perform inference performance. All in all, we have run trials with a single *GRU* layer, 2 stacked layers of *GRU* cells, with and without self attention and using several numbers of neurons and *GRU* cells.

On our initial exploration step we have used early stopping in *Keras*, which prevents a model from overfitting by watching the loss on the holdout set. Essentially, training is halted when the model starts to lose its generalization ability by miss-predicting the validation set. A good approach to this, is to use a broad starting network and evaluate how and if additional epochs help the model predict the validation set.

To summarize, we hand-coded a few baseline *RNN* architectures, ran them in sequence and kept the best model, based on the *f1 score*, which we then used for performing hyperparameter tuning. This procedure allowed us for eyeballing some of the parameters that might influence the performance thereby cutting down on the number of parameters that we should perform hyperparameter tuning on, given that it is an expensive procedure.

3.1 Recurrent Units

3.1.1 Gated Recurrent Unit (*GRU*) cells

For completeness, we present a summary of the internal architecture of a *GRU* cell below:

- **Update gate**: The update gate acts similar to the forget and input gate of an *LSTM*. It decides what historical information to keep and discard.
- **Reset gate**: The reset gate decides how much past information to forget.

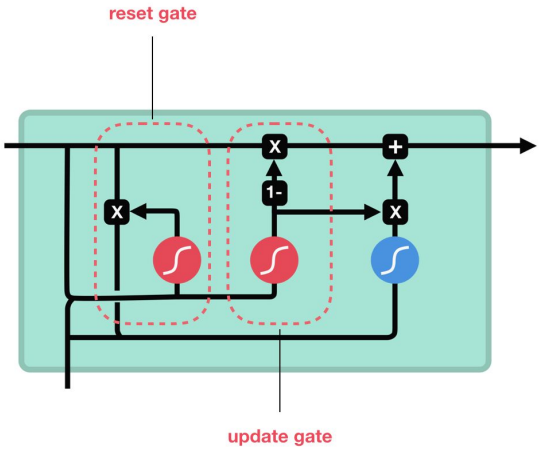


Fig. 3.1.1: The *GRU* cell architecture
(source: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21)

3.1.2 Long-Short Term Memory (*LSTM*) cells

Having experimented with *GRU* cells, we proceeded with an architecture based on *LSTM* cells.

We present a summary of the internal architecture of a *LSTM* cell below:

- **Forget gate:** this gate decides what information should be discarded or kept to influence subsequent calculations. Information from the previous hidden state and information from the current input is passed through the sigmoid function. Values come out between 0 and 1. Closer to 0 indicated we should discard history, closer to 1 indicates we should keep history.
- **Input Gate:** this gate updates the cell state. Previous hidden state and current input is passed to the sigmoid function, thereby deciding what values should be updated by examining the sigmoid output. Values closer to 0 are not important, values closer to 1 are important. The values are normalized between -1 and 1 with the tanh function, the output of which is multiplied with the sigmoid output. The sigmoid output will decide which information is important to keep from the tanh output.
- **Cell State:** the cell state is pointwise multiplied with the forget vector. Cell state values close to 0 are dropped. Subsequently, new cell state is acquired by adding the output of the input gate.
- **Output Gate:** this gate decides what the next hidden state should be. Previous hidden state and current input is passed to the sigmoid function, new state is passed to the tanh function, their outputs are multiplied so as to decide what information the hidden state should carry. The output is the hidden state. The new cell state and the new hidden is then carried over to the next time step.

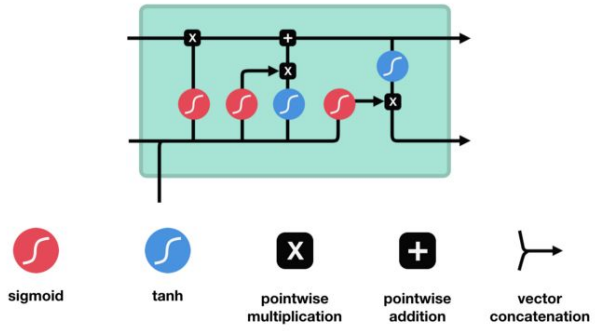


Fig. 3.1.2: *LSTM* cell architecture
(source: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21)

3.2 OBSERVATIONS

In addition to the previous observations we had trying to train a good MLP model, in this task we decided to play more with other parameters more targeted to *RNN*.

3.2.1 NEURON SIZE & REGULARIZATION

Neuron size, along with epochs and regularization techniques, are crucial in terms of the bias-variance trade-off. Using a small neuron size would lead to underfitting while an unnecessarily big size to overfitting. As for regularization we used the already mentioned early stopping and dropout methods. Dropout would produce varying degrees of results depending on the rate used. A large rate will introduce more bias to the model and possibly lead to underfitting while a low rate will simply reduce variance a little. Our baseline model seemed to

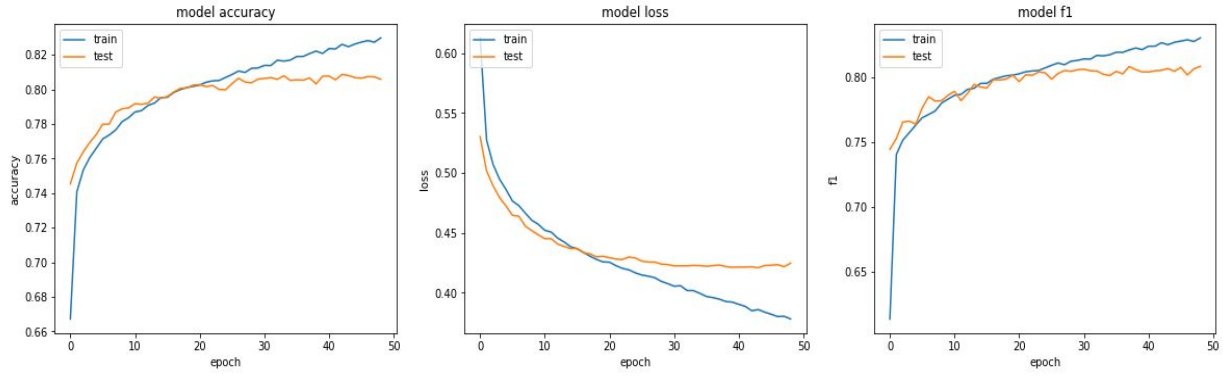
work better for values in the [0.2, 0.5] range which could mean that we have enough bias in our model and need some variance reduction.

3.2.2 OPTIMIZERS

We tried *SGD* and *Adam* as our optimization strategies, but in the end, the optimizer of our choice was *Adam*, the main reason for that is that it had almost the same results as *SGD* and it wouldn't make much sense to use both of them on our tuning stage.

3.2.3 CONCLUSION

The best base model was selected based mainly on the performance of the model (*f1 score*) on the dev set. In order to decide, we observed all manual executions of the models and selected the one with the best *f1 score*. The best model's graph is shown below:



In general, we observed that throughout our experiments the validation *f1* score was beginning to decrease after 20-22 epochs. A complete picture of the iterations is present in our *Jupyter Notebook*.

```
Test binary cross entropy: 0.4258

Test precision: 0.7883

Test recall: 0.8399

Test f1: 0.8117

Test accuracy: 0.8070
```

Moreover, using 2 layers of *GRU* cells did not improve our validation *f1 score*. The model is already able to overfit on the training data after 20 epochs and did not predict the test set correctly. The only way we could improve our score is trying to find more features to describe our twits better.

Furthermore, we spend some time to decide which layer would help us with tracking long-term dependencies effectively while mitigating the vanishing/exploding gradient problems. Given infinite computational power we would effectively test all possible options, but in general *LSTM* cells have not proven significantly better results in comparison to *GRU*, and given that it took us around double the time to train an *LSTM*, we concluded that for the current task of classifying tweets, the above *RNN* with *GRU* (50 units) layer and 1 Dense layer (50 neurons) was more than enough for our task.

4. HYPERPARAMETER OPTIMIZATION

During the hyperparameter tuning phase we considered several options. Some of the most widely used approaches have been mentioned in the context of the previous assignment, we reiterate the most common briefly here:

- *Baby-sitting (Trial & Error)*
- *Grid Search*
- *Random Search*
- *Bayesian Optimization*

4.1 IMPLEMENTATION DETAILS

In order to search our hyperparameter space, we opted for using *Talos*, an open-source hyperparameter optimization package known to be working well with *Keras*.

4.2 METHODOLOGY

After having established a base model, we performed hyperparameter tuning with *Talos* on the selected model. The selection criterion was based on the *Talos* automatic best-model selection mechanism, which employees the *f1 score*. From our starting tests we have seen that the best performing model was created using single layer *GRU* cells and and an *MLP* of 50 neurons each which got 0.8117 *f1 score* . So on our tuning we chose to tune the *MLP* with *GRU* cells only. The most important reason for that was that it took almost 36 hours with *Talos* to tune for 48 cases, which would be doubled if we also used *LSTM* cells too.

We chose to search for the following mix:

- 1 layer of *GRU* cells.
- Different size of memory for the *GRU* layer [100, 200, 300]
- Different size of neurons on the dense layer after *GRU* [16, 32, 50, 64, 100]
- Training for 50 or 100 epochs (in most cases, early stopping prevented iteration through all epochs)
- Recurrent dropout of 0.2 or 0.5 for the *GRU* cells
- Dropout rate of 0.2, or 0.5 for the *MLP* layer

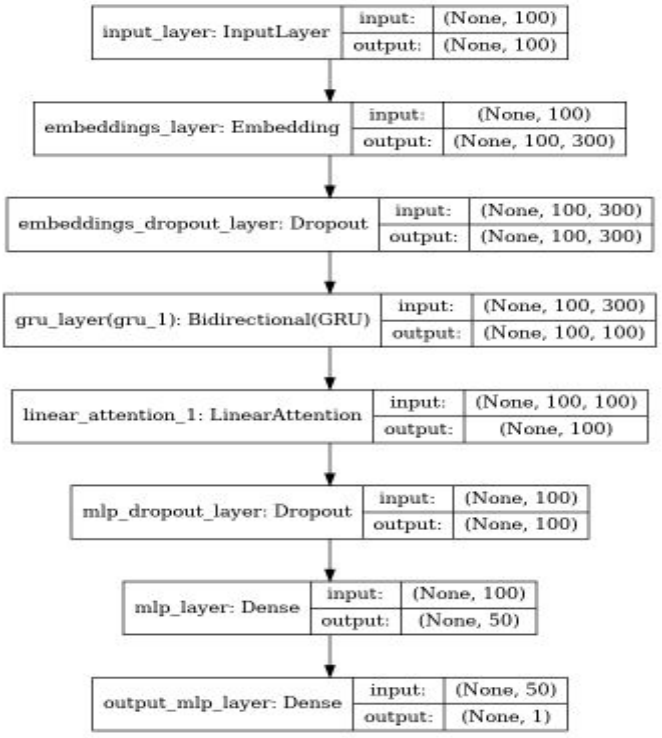
The hyperparameters search space is in *Python* dictionary format as required by *Talos*, is shown below:

```
hyperparameters = {
    'learning_rate': [1e-3],
    'embeddings_dropout': [0.2],
    'gru_size': [100, 200, 300, 500],
    'gru_recurrent_dropout': [0.2],
    'gru_dropout':[0.2],
    'mlp_num_neurons': [32, 50, 64, 100, 500],
    'mlp_activation': ['relu'],
    'attention': ['linear', 'deep'],
    'optimizer': [Adam],
```

```
        'output_num_neurons': [1],
        'output_layer_activation': ['sigmoid'],
        'epochs': [50]
    }
```

5. RESULTS

The final architecture of our *RNN* is shown below:



Before using the *held-out* dataset we normalized it, meaning that we applied the same preprocess as with the training data. The resulting *f1 score* was 0.82 and the confusion matrix of the unseen dataset was the below:

	precision	recall	f1-score	support
0	0.84	0.79	0.81	177
1	0.80	0.85	0.83	182
accuracy			0.82	359
macro avg	0.82	0.82	0.82	359
weighted avg	0.82	0.82	0.82	359

The final classification gives a good ratio between the *TN-FN* (5/2) and *TP-FP* (5/1) so we have an overall efficient classifier. Some random examples of misclassification are depicted below:

	actual	predicted	text
26	0	1	rt new time warner slogan time warner where we make you long for the days before cable
27	0	1	i still love my kindle2 but reading the new york times on it does not feel natural i miss the bloomingdale ads
28	0	1	although today s keynote rocked for every great announcement at amp t shit on us just a little bit more
29	0	1	fuzzball is more fun than at amp t p ball com twitter
36	1	0	lebron best athlete of our generation if not all time basketball related i don t want to get into inter sport debates about $\frac{}{2}$
37	1	0	lebron is a beast nobody in the nba comes even close
38	1	0	good news just had a call from the visa office saying everything is fine what a relief i am sick of scams out there stealing
39	1	0	rt i love the nerdy stanford human biology videos makes me miss school

From the above results we can see that the tweets that our classifier failed to predict correctly are very obscure and are really hard even for a human to understand the correct sentiment. (We have the full list of misclassified tweets in the notebook).